POLITECNICO
MILANO 1863

Enabling the High Level Synthesis of Data Analytics Accelerators

Marco Minutoli, Vito Giovanni Castellana, Antonino Tumeo, Marco Lattuada, Fabrizio Ferrandi

Marco Minutoli, Vito Giovanni Castellana, Antonino Tumeo, Marco Lattuada, and Fabrizio Ferrandi. Enabling the high level synthesis of data analytics accelerators. In *Proceedings of the Eleventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES '16, pages 15:1–15:3, New York, NY, USA, 2016. ACM

The final pubblication is available via http://dx.doi.org/10.1145/2968456.2976764

Enabling the High Level Synthesis of Data Analytics Accelerators

Marco Minutoli, Vito Giovanni Castellana, Antonino Tumeo
High Performance Computing
Pacific Northwest National Laboratory
99352 Richland, WA, USA
{marco.minutoli, vitoGiovanni.castellana, antonino.tumeo}@pnnl.gov

Marco Lattuada, Fabrizio Ferrandi
Dipartimento di Elettronica, Informazione e Bioingegneria
Politecnico di Milano
20132 Milano, Italy
{marco.lattuada, fabrizio.ferrandi}@polimi.it

ABSTRACT

Conventional High Level Synthesis (HLS) tools mainly target compute intensive kernels typical of digital signal processing applications. We are developing techniques and architectural templates to enable HLS of data analytics applications. These applications are memory intensive, present fine-grained, unpredictable data accesses, and irregular, dynamic task parallelism. We discuss an architectural template based around a distributed controller to efficiently exploit thread level parallelism. We present a memory interface that supports parallel memory subsystems and enables implementing atomic memory operations. We introduce a dynamic task scheduling approach to efficiently execute heavily unbalanced workload. The templates are validated by synthesizing queries from the Lehigh University Benchmark (LUBM), a well know SPARQL benchmark.

1. INTRODUCTION

Data Analytics applications, such as graph databases, often employ pointer or linked list-based data structures that, although convenient to represents dynamically changing relationships among the data elements, induce an irregular behavior [7]. These data structures, in fact, allows spawning many concurrent activities, but present many unpredictable, fine-grained, data accesses, and require many synchronization operations. Partitioning the datasets without generating load imbalance is also very difficult. Conventional general-purpose architectures are optimized for locality and reduced access latency, and do not cope well with these workloads, making application-specific accelerators (implemented, for example, on Field Programmable Gate Arrays - FPGAs) an appealing solution [6]. However, conventional High Level Synthesis (HLS) flows traditionally perform well with compute intensive workloads (i.e., digital signal processing) that mainly expose instruction level parallelism, and can be easily partitioned across replicated functional units. They also usually assume a simple memory system focusing more to reduce, rather than tolerate, access latencies.

Resource Description Framework (RDF) databases have become one of the most prominent example of data analytics application. RDF is the metadata data model typically used to describe the Semantic Web. RDF databases naturally maps to graphs, and query languages for these databases such as SPARQL basically express queries as a combination of graph methods (graph walks and graph pattern matching operations) and analytic functions. Among these, we consider GEMS, the Graph database Engine for Multithreaded Systems (GEMS) [2]. GEMS implements a RDF database on a commodity cluster by mainly employing graph methods at all levels of his stack. To address the limitations of HPC systems, GEMS employs a runtime (Global Memory and threading - GMT) that provides: a global address space across the cluster, so that data do not need to be partitioned, lightweight software multithreading, to tolerate data access latencies, and message aggregation, to improve network utilization with fine-grained transactions. A graph application programming interface (API) and a set of methods to ingest RDF triples and generate the related graph and dictionary (collectively named SGLib) are built with the functions provided by the runtime. On top of the whole system, a translator converts query expressed in SPARQL to graph-pattern matching operations in C/C++.

We have investigated acceleration of queries, as generated for the GEMS software stack, on FPGAs. We have developed a set of architectural templates and HLS methodologies to automatically generate specifications in hardware description language (Verilog) of graph methods starting from C descriptions and have integrated them in a modified version of an openly available High Level Synthesis tool, Bambu [1]. We have then interfaced GEMS with the modified Bambu, and generated accelerators for SPARQL queries coming from the Lehigh University Benchmark (LUBM) [5], a reference benchmarks for Semantic Web Repositories.

2. ARCHITECTURE OVERVIEW

Figure 1 shows how the GEMS stack has been integrated with Bambu. We have changed GEMS layer so that they are not anymore dependent on the GMT runtime. We developed a pure C version of the graph API that does not exploit any of the functionalities of GMT. We modified the code emitter accordingly, so that the C code generated from the SPARQL queries only invokes the new C graph API to perform graph

DOI: http://dx.doi.org/10.1145/0000000.0000000

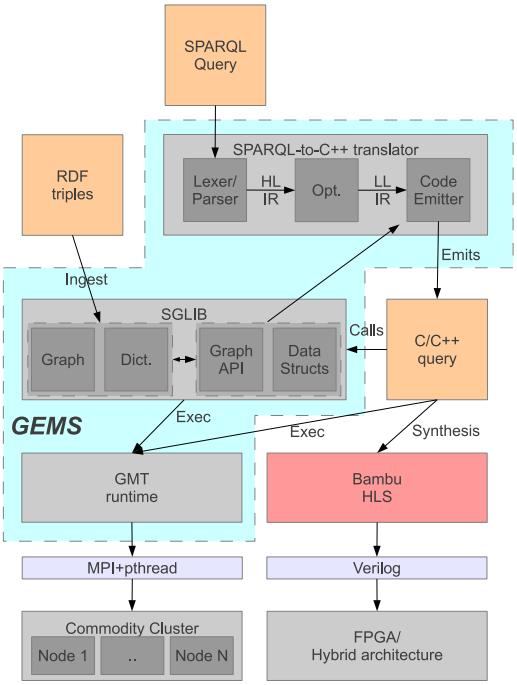

Figure 1: Structure of the GEMS stack and interaction with Bambu HLS

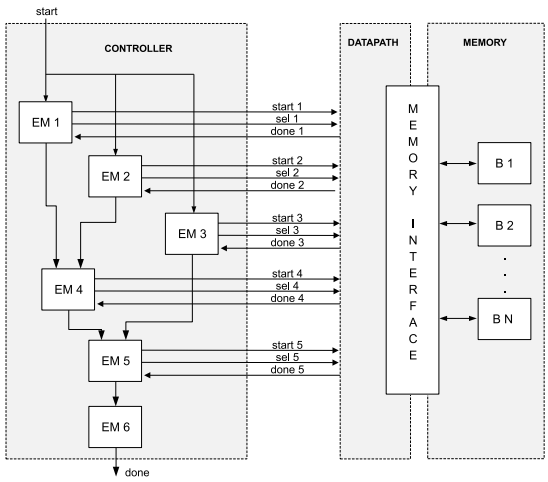

Figure 2: High Level overview of an architecture combining the parallel controller and the hierarchical memory interface.

walks and matching and pure C functions to execute any analytic operation. Such a C code in practice corresponds to a set of nested loops, where each loop matches a particular edge of the graph pattern that composes the query, and becomes the input of Bambu for the synthesis. Note that synthesizing full queries is not a limitation: in the many analytics applications, once the query is defined, it remains stable in time, while the dataset dynamically changes. So, provided that the query execution provides a speed up, the time required to synthesize the query on FPGA is affordable.

Bambu has been modified by progressively integrating three architectural templates, and the corresponding methodologies to generate the instances of the templates. The three components aim at providing better support for certain of the typical structures and behaviors that characterize parallel graph methods. These obviously include the graph pattern matching methods employed in GEMS for the query processing. The components include a Parallel distributed Controller (PC) [3], a Hierarchical, multi-ported, Memory Interface (HMI) [4], and a Dynamic Task Scheduler (DTS).

The PC allows to generate more efficient designs that exploit coarse grained (task level) parallelism than the typical centralized controllers of conventional HLS flows based around the Finite State Machine with Datapath (FSMD) model, in terms of performance and area utilization. This is a key element in accelerating graph algorithms that basically are composed of a varying number of nested loops, iterating on vertices or edges, where each iteration could identify a different task. By adopting the PC, it is easy to coordinate parallel execution of tasks (one, or more iterations each) on an array of replicated accelerators. The PC consists of a set of communicating modules, each one associated with one, or more, operations. Controller modules are called Execution Managers (EMs). EMs start execution of the associated operations as soon as all their dependencies are satisfied and resource conflicts resolved. Other dedicated components, named Resource Managers (RM), arbitrate execution of operations on shared resources. EMs communicate through a token-based schema: when a dependency is satisfied, a token is passed to the next EM. When all dependency for an EM are satisfied, it checks RMs for resource availability and, if the resource is free, execution starts.

The HMI provides an easy way to dynamically disambiguate fine grained memory accesses to locations of a large, multi-banked memory, while maintaining to the HLS flow and the accelerator pool the abstract view of a shared memory. In cooperation with the PC, the HMI also enables to easily support atomic memory operations. Graph algorithms typically access unpredictable memory locations with fine-grained transactions (i.e., the follow pointers), and their implementation is much easier when considering a shared memory abstraction. Also, they often are synchronization intensive when parallelized, because the different tasks may access the same elements concurrently. The HMI takes in input memory access requests from N ports, which have an address, a data and an operation type (load/store) line. The MIC routes requests towards one of the B output ports, corresponding to a different memory bank, by evaluating their addresses. Each memory bank has non-overlapping addresses. Accesses are routed towards a specific memory port at runtime, providing efficient support of the unpredictable memory access patterns typical in irregular applications. Furthermore, address can be scrambled across banks (i.e., consecutive addresses are interleaved on different banks) to reduce hotspots. Control logic, synthesized according to the specific scrambling function, performs the routing. Support to atomic operations is provided through a mechanism similar to the RMs, as access to the specific memory port is not granted until termination of the atomic operation. Figure 2 provides a combined view of the PC and the HMI. RMs are hidden in the datapath (at the front of shared resources) and in the memory interface (at front of the memory ports).

The DTS provides a way to execute new tasks as soon as one of the multiple accelerators is free: instead of simply

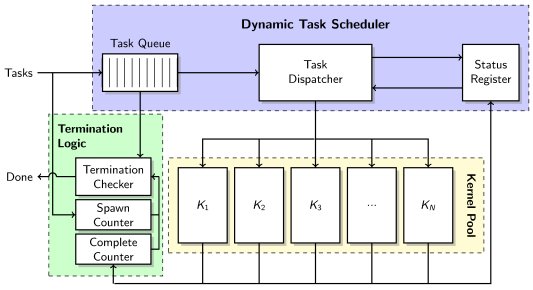

Figure 3: High level overview of an architecture template using the DTS

using a fork/join model, where all currently executing tasks on the set of accelerators must terminate before a new group could be executed, the DTS allow scheduling new tasks as soon as one of the accelerators is free. This adapt to a variety of graph algorithms where certain tasks (iterations) may execute for a long time, while other could terminate early, such as when a graph walk is pruned early because it reached an uninteresting part of the graph. Figure 3 shows an architecture template integrating the DTS. The DTS interfaces to the set of parallel accelerators (Kernel Pool) and to the Termination Logic. The DTS itself contains a Task Queue, the Task Dispatcher, and a Status Register that holds information on the accelerators in the kernel pool. As soon as a task is terminates, the status register is updated and the DTS can schedule a new task. The Termination Logic allows understanding when all the tasks have completed, i.e., when for example all iterations of a parallel loop have completed.

3. RESULTS OVERVIEW

To validate our architectural templates and our approach, we have we have synthesized 7 queries from LUBM, and we have tested the performance using a dataset of 5,309,056 RDF "triples". In Table 1, we compare, in terms of execution latency, a serial implementation of the architecture (Single Acc.), one that employs PC and the HMI [3] (Parallel Controller), and one that also includes the DTS (Dynamic Scheduler). The parallel architectures include 4 accelerators and HMIs with 4 ports. With respect to the serial implementation, the architectures employing the DTS generally show a speed up close to the theoretical maximum. In many cases, the DTS also provides significant speed ups against the PC designs. This happens, in particular, with queries that have some iterations (tasks) that executes order of magnitudes longer than others. Another important effect of the DTS is that it maximizes utilization of the available memory channels as provided by the HMI. In fact, all the architectures with the DTS utilize at least 3 out of 4 of the memory ports for more than 75% of the time.

4. CONCLUSIONS

We have been investigating architectural templates and methodologies to enable the HLS of data analytics applications. Among all data analytics applications, we focused our attention to RDF databases. These operate on large amount of data that can be conveniently organized in graph data structures and queried through languages that express queries as graph pattern matching operations. Modern reconfigurable devices appear a promising target to accelerate this type of applications, which are massively parallel and mainly memory bound. However, conventional HLS flows typically target digital signal processing applications, which usually are compute intensive, exhibit significant instruction level parallelism, and present regular memory access patterns. They do not cope well with the fine-grained, unpredictable memory accesses, and the highly dynamic, and sometimes very unbalanced, coarse grained (task) parallelism, typical of graph methods. We detailed three of the architectural templates and the related methodologies we have been designing to address these issues, and presented some initial results on a realistic benchmark. We believe that our effort can provide a solid basis to make HLS of this type of applications more viable, thus a productive way for the data analytic community to use custom accelerators.

Table 1: Performance comparison of the implementation using the DTS+HMI against the serial and the parallel controller implementations

	Single Acc. # Cycles	Parallel Controller # Cycles	Dynamic Scheduler # Cycles	Speedup Single Acc.	Speedup Parallel Controller
Q1	1,082,526,974	1,001,581,548	287,527,463	3.76	3.48
Q2	7,359,732	2,801,694	2,672,295	2.75	1.05
Q3	308,586,247	98,163,298	95,154,310	3.24	1.03
Q4	63,825	42,279	19,890	3.21	2.13
Q5	33,322	13,400	8,992	3.71	1.49
Q6	682,949	629,671	199,749	3.42	3.15
Q7	85,341,784	35,511,299	24,430,557	3.49	1.45

5. REFERENCES

[1] Bambu: A Free Framework for the High-Level Synthesis of Complex Applications. http://panda.dei.polimi.it, 2014.

[2] V. Castellana, A. Morari, J. Weaver, A. Tumeo, D. Haglin, O. Villa, and J. Feo. In-memory graph databases for web-scale data. *Computer*, 48(3):24–35, Mar 2015.

[3] V. G. Castellana, M. Minutoli, A. Morari, A. Tumeo, M. Lattuada, and F. Ferrandi. High Level Synthesis of RDF Queries for Graph Analytics. In *ICCAD'15: IEEE/ACM International Conference on Computer-Aided Design*, pages 323–330, 2015.

[4] V. G. Castellana, A. Tumeo, and F. Ferrandi. An adaptive memory interface controller for improving bandwidth utilization of hybrid and reconfigurable systems. In *DATE 2014: Design, Automation and Test in Europe*, pages 1–4, 2014.

[5] Y. Guo, Z. Pan, and J. Heflin. Lubm: A Benchmark for OWL Knowledge Base Systems. *Web Semant.*, 3(2-3):158–182, Oct. 2005.

[6] A. Putnam, A. Caulfield, E. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Xiao, and D. Burger. A reconfigurable fabric for accelerating large-scale datacenter services. In *ISCA: ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 13–24, June 2014.

[7] A. Tumeo and J. Feo. Irregular applications: From architectures to algorithms [guest editors' introduction]. *IEEE Computer*, 48(8):14–16, 2015.